

Performance Evaluation of Brokerless Messaging Libraries

Lorenzo La Corte
Hitachi Energy Research
Baden-Dättwil, Switzerland

Syed Aftab Rashid
Hitachi Energy Research
Baden-Dättwil, Switzerland

Andrei-Marian Dan
Hitachi Energy Research
Baden-Dättwil, Switzerland

Abstract—Messaging systems are essential for efficiently transferring large volumes of data, ensuring rapid response times and high-throughput communication. The state-of-the-art on messaging systems mainly focuses on the performance evaluation of *brokered* messaging systems, which use an intermediate broker to guarantee reliability and quality of service. However, over the past decade, *brokerless* messaging systems have emerged, eliminating the single point of failure and trading off reliability guarantees for higher performance. Still, the state-of-the-art on evaluating the performance of brokerless systems is scarce. In this work, we solely focus on brokerless messaging systems. First, we perform a qualitative analysis of several possible candidates, to find the most promising ones. We then design and implement an extensive open-source benchmarking suite to systematically and fairly evaluate the performance of the chosen libraries, namely, ZeroMQ, NanoMsg, and NanoMsg-Next-Generation (NNG). We evaluate these libraries considering different metrics and workload conditions, and provide useful insights into their limitations. Our analysis enables practitioners to select the most suitable library for their requirements.

Index Terms—Messaging, Publish/Subscribe, Message queue, ZeroMQ, NanoMsg, NNG, Brokerless messaging library

I. INTRODUCTION

Efficiently transferring large amounts of data while ensuring quick response times is a significant challenge in control systems, Internet-of-Things (IoT), robotics, computer networks, etc. *Messaging libraries* enable the exchange of messages/packets between various software components, offering high throughput and low latency.

Messaging libraries or frameworks typically use message queues to handle *asynchronous* communication, ensuring reliable message transfer even when some components are temporarily unavailable. Additionally, most systems include a *broker*, an intermediary service that routes and distributes messages between producers and consumers. Brokered systems usually offer features such as message persistence, guaranteed delivery, and built-in load balancing.

Brokerless architectures eliminate the broker, removing the single point of failure and enabling publishers and subscribers to communicate directly, thus achieving higher performance in terms of latency [1] and bandwidth [2]. However, this shifts responsibilities for reliability, discovery, and routing to the application layer, which in turn offers greater flexibility by allowing developers to tailor these aspects precisely to their requirements. Moreover, brokerless libraries are typically lightweight, with a small memory footprint and minimal

dependencies, making them particularly suited for resource-constrained environments [3], [4]. Therefore, brokerless frameworks have emerged in various contexts, such as control and monitoring systems [4], IoT [5], and robotics [6]–[8].

Several state-of-the-art studies focus on evaluating the performance of brokered systems [5], [9]–[12], considering performance metrics such as throughput, latency, CPU, and memory consumption. However, to the best of our knowledge, only a handful of works in the state-of-the-art include brokerless solutions in their analysis [1]–[3], and most of their results are outdated considering the rapidly evolving landscape of messaging systems. Therefore, new insights on the performance capabilities and limitations of brokerless messaging frameworks are necessary.

In this work, we address the gap in the state-of-the-art by analyzing several brokerless libraries qualitatively and quantitatively. Specifically, this work makes the following contributions:

- 1) A qualitative analysis of existing brokerless messaging libraries, focusing on licensing, ease of installation, documentation, maintenance, and community support. Based on the qualitative analysis, we choose three libraries, namely ZeroMQ, NanoMsg, and NNG, for performance evaluations.
- 2) An open-source benchmarking suite [13] to systematically and fairly evaluate the performance of the three chosen libraries under different settings. The benchmarking suite can be easily extended to include other libraries.
- 3) A thorough experimental evaluation to compare the performance of the chosen libraries, focusing on latency, throughput, jitter, CPU usage, and memory consumption. The results obtained show the trade-offs between the libraries in various scenarios, offering insights into their optimal use cases.

The article is organized as follows. Section II outlines the general characteristics of messaging libraries. Section III presents a qualitative analysis of several brokerless messaging frameworks. Section IV describes our benchmarking suite, designed to fairly and systematically compare the performance of the selected libraries. Section V discusses the experimental results and the insights derived from them. Finally, Section VI reviews related work and Section VII concludes the article.

II. OVERVIEW OF MESSAGING LIBRARIES

In this section, we discuss communication transports and patterns commonly offered by messaging libraries. Then, we proceed by highlighting key differences between brokered and brokerless architectures.

Messaging libraries typically rely on three transport mechanisms: 1) the *Transmission Control Protocol* (TCP) is primarily used for network communications and leverages the operating system's network stack to ensure reliable data transfer. 2) *Inter-Process communication* (IPC) mechanisms, such as Unix domain sockets, allow two processes on the same machine to exchange data without traversing the network stack, through file-system paths or unique local identifiers, generally incurring into lower latencies compared to the TCP communication. 3) *In-Process communication* enables threads in the same process to exchange data directly via shared memory, and is therefore the fastest option, as it avoids the overhead of the operating-system networking stack, significantly reducing communication latencies.

On top of the transport, messaging frameworks offer standardized communication patterns. *Request/Reply* (REQ/REP) is a message-centric pattern, used for Remote-Procedure-Call (RPC)-like interactions, where a message is sent as a request, with a corresponding reply expected from the receiver. *Publish/Subscribe* (PUB/SUB) is a data-centric pattern, where publishers broadcast messages to interested subscribers, registered to specific topics. This provides space, time, and synchronization decoupling [14].

Traditional Message-Oriented Middleware (MOM) platforms — such as Apache Kafka [15] and RabbitMQ [16] — use a *message broker*, a long-running server responsible for topic management, persistence, and routing. The broker decouples producers and consumers in *space* (they need not know each other) and *time* (messages can be stored until a consumer reconnects), enabling different quality of service guarantees. These features come at the price of (i) an extra network hop (store-and-forward), and (ii) a single logical point of failure. On the other hand, brokerless libraries omit the intermediary broker entirely. They provide enhanced socket abstractions that enable *direct communication* among peers. Messages are byte sequences that the library queues internally between threads or processes and exposes at the destination socket. As a result, latency is reduced to a single network hop and can approach the transport's theoretical minimum, whereas persistence, retransmission, and higher-level delivery guarantees must be implemented at the application layer, when required.

III. QUALITATIVE ANALYSIS

In this section, we discuss five brokerless messaging libraries: ZeroMQ, NanoMsg, NNG, ZeroRPC, and YAMI4. We qualitatively assess them based on license, documentation, and community support.

Table I reports technical facts and practical aspects of the libraries considered. For all, installation is straightforward, and is therefore omitted. ZeroRPC offers little up-to-date

documentation or active community support, and YAMI4 lacks a public code repository. By contrast, the activity counters show that NNG is the only library currently under active development.

In the following, we provide an overview of the candidates.

1) *ZeroMQ*: [29] is a C++ library known for its scalability and ease of use in building high-throughput applications. The project started in 2007 as an alternative to the Advanced Message Queuing Protocol (AMQP) and is currently one of the most popular brokerless messaging libraries. It supports data-centric communication through its PUB/SUB model, which includes advanced patterns for performance, reliability, state distribution, and monitoring. The library offers a unified socket API that abstracts over transport details, simplifying application development by decoupling messaging logic from the underlying communication layer. In particular, core operations — bind, connect, send, and receive — are transport-agnostic, with the user specifying the transport via the endpoint address.

2) *NanoMsg*: [30] is a re-imagining of ZeroMQ, realized as a complete rewrite in C, focusing on a lower memory footprint and improved algorithm efficiency [31]. NanoMsg keeps a simple, transport-agnostic socket API, while aiming to improve upon several design aspects of ZeroMQ — particularly by targeting full POSIX compliance to facilitate development and maintenance. It is not in active development but only in sustaining mode.

3) *NNG*: NNG [32] builds upon the design principles of ZeroMQ and NanoMsg while addressing many of their limitations, as detailed in its rationale [33]. It has a new implementation, focusing on scalability, portability, and robustness. Its API removes legacy constraints from POSIX socket interfaces, offering a cleaner and more portable abstraction and facilitating the development of new transports. NNG eliminates the complex, error-prone state machine architecture of NanoMsg and relaxes its file-descriptor-driven implementation, aiming to overcome the poor scalability of NanoMsg's threading model. To this end, NNG performs all I/O operations asynchronously, which also simplifies integration with non-blocking and event-driven applications.

4) *ZeroRPC*: [34] extends the capabilities of ZeroMQ by providing integrated RPC support. Developed in Python, ZeroRPC offers a dynamically typed service interface that simplifies the construction of service-oriented architectures. It leverages MessagePack for efficient serialization of data in a JSON-compatible format. However, it only supports data-centric communication through a simplified PUB/SUB pattern. Additionally, the setup and usage might be more challenging due to the lack of clear documentation.

5) *YAMI4*: [24] is a lightweight messaging framework, designed for real-time control and monitoring applications. It emphasizes simplicity in message exchange and offers features suited for resource-constrained environments. It provides a simple form of PUB/SUB via its value publisher concept, and it is limited to In-Process and TCP transports. Unlike other frameworks, YAMI4 distinguishes between physical connections and logical destinations, allowing different com-

TABLE I: Qualitative characteristics of the considered brokerless messaging libraries.

Characteristic	ZeroMQ	NanoMsg	NNG	ZeroRPC	YAMI4
Development language	C++	C	C	Python	C++, Objective C
Year of first release	2007	2012	2016	2012	2010
Client bindings	C, C++, ...	C, C++, ...	C	Python, node.js	C++, ...
License	MPL-2.0 [17]	MIT [18]	MIT [19]	MIT [20]	BSL or GPL
Documentation	Extensive [21]	Good [22]	Good [23]	Limited	Extensive [24]
GitHub Stars / Forks / Issues	10.1k / 2.4k / 1.5k	6.1k / 1k / 600	4k / 500 / 1.1k	3.2k / 300 / 100	—
GitHub Recent Commits / Issues	0 / 20	0 / 2	100 / 18	2 / 0	—

Notes: GitHub metrics were retrieved in April 2025. Commit and issue activity covers January–April 2025. Counts are rounded and intended as “at least”. GitHub activity considered for **ZeroMQ** refers specifically to `zeromq/libzmq` [25]. For **NanoMsg**, it refers to `nanomsg/nanomsg` [26]. For **NNG**, it refers to `nanomsg/nng` [27]. For **ZeroRPC**, it refers to `0rpc/zerorpc-python` [28].

TABLE II: Versions and descriptions of the libraries.

Library	Version and Description
ZeroMQ	libzmq (core API, v4.3.4), CZMQ (High-level C Binding, v4.2.1)
NanoMsg	libnanomsg (C core API, v1.1.5)
NNG	libnng (C core API, v1.9.0)

munication patterns to coexist over a single connection and avoiding early commitment to a specific pattern [35]. YAMI4’s source code is available under a GPL license, with commercial licensing options upon request, but it *does not have a public repository*, which may restrict broader community adoption.

We exclude YAMI4 and ZeroRPC from our quantitative analysis, as YAMI4 lacks support for Inter-Process communication and does not provide a public repository, whereas ZeroRPC is primarily focused on remote procedure calls. Therefore, we proceed with a detailed analysis of ZeroMQ, NanoMsg, and NNG, as they (i) provide a similar, simple, and efficient API, (ii) are actively maintained, and (iii) with a large community support.

IV. METHODOLOGY

This section introduces the design of our benchmarking suite, developed to systematically compare the performance of messaging libraries. We describe the benchmarking experiment and the architecture of our testing framework, designed to ensure a fair and unbiased comparison of library performance. Additionally, we outline the workload conditions and define the performance metrics considered in our evaluation.

We consider the communication transports described in Section II, namely *In-Process*, *Inter-Process*, and *TCP*. The focus is on the PUB/SUB communication, fixing a single, representative pattern — the most common we have observed in practice for messaging frameworks. We consider the ZeroMQ, NanoMsg, and NNG libraries, as motivated in Section III. Library versions used are described in Table II.

A. Parameters

For our benchmark, we consider the following parameters:

- The *number of subscribers* consuming the messages from the queue.

- The *number of messages sent* from the publisher to all the subscribers.
- The *publishing interval*, i.e., the amount of microseconds between each sending of a message.
- The *payload size*, i.e., the size of each message.
- The *publishing delay*, i.e., the time in milliseconds that the publisher waits before starting to publish messages.

These parameters are passed as arguments to the simulator, together with the transport and the library used for the communication.

B. Figures of Merit

We consider latency, throughput, jitter, and CPU and memory usage as the figures of merit for our comparison.

- *Latency* measures the delay of the system, crucial in real-time and streaming applications, and provides insight into the responsiveness of the communication framework.
- *Throughput* represents the rate at which messages are successfully delivered, reflecting the efficiency of the system under various workload conditions.
- *CPU and memory usage* reveals the resource cost associated with achieving the observed performance, a tradeoff especially relevant in systems with tight resource budgets or real-time constraints.
- *Jitter* measures the variation in latency over time. This metric is important in real-time applications, as high jitter can lead to unpredictable behavior in systems that rely on timely message delivery.

For M messages successfully received, we consider the overall distribution of *one-way transmission latencies* l_i , with $i \in [1, M]$. To compute the latency, we encode a UNIX timestamp in the payload of each message sent. If the message is received, we compute the latency as the difference between the receiver and the sender timestamps. Finally, upon completion of the simulation, we compute *minimum*, *average*, $p90$, $p99$, and *maximum* latencies.

Secondly, we consider *throughput* as the number of megabytes delivered per second. We compute it considering the number of messages received M , which all have the same fixed size \mathcal{P} , over the amount of seconds that passed from the first message sent to the last message received.

Then, we consider *the average one-way packet delay variation*, introduced in the RFC 3393 IETF standard track

document [36], here defined as “jitter”, and computed as the average of the absolute difference between consecutive transmission latencies:

$$\text{jitter} = \frac{1}{M-1} \sum_{i=1}^{M-1} |l_i - l_{i-1}|. \quad (1)$$

Finally, we consider *median CPU* and *median memory* usage percentages. To compute these metrics, we take discrete samples throughout the simulation and calculate their median values. We leverage the `psutil` Python package [37] to sample these metrics at each time step. For memory usage, we specifically measure the *Unique Set Size (USS)*, which represents the amount of memory unique to a process and not shared with others [38]. To account for multiple threads or processes running simultaneously, we *aggregate* CPU and memory usage across all processes at each time step. Specifically, at each sampling time t , we sum the percentages for the CPU and USS memory usage over all N processes. Then, we report the median of the aggregated values as the final metric.

C. Benchmarking Experiment

In our experiments, we use one publisher and S subscribers. The subscribers run first, whereas the publisher is executed after a delay of \mathcal{D} milliseconds. The publisher enqueues a total of \mathcal{C} messages, each intended to be received by all subscribers. However, due to potential losses, $M \leq \mathcal{C}$ denotes the actual number of messages successfully received. Each message carries a payload of \mathcal{P} bytes. A timestamp for latency measurement is embedded in the message, and padded to the required size, thus introducing no extra overhead. The publisher sends a message every \mathcal{T} microseconds (publishing time interval). When $\mathcal{T} = 0$, the publisher sends messages at the maximum achievable rate, without delay. Algorithm 1 summarizes the behavior of the publisher and subscribers.

D. Benchmarking Suite

Our framework is designed to be general and extensible, offering a fair and comprehensive evaluation of library performance across different workload conditions. Figure 1 shows a schematic overview of our framework components.

We conduct an extensive simulation, covering sets of the parameters detailed in Section IV-A. All possible combinations of the parameter values are systematically explored, resulting in a total of $\prod_{i=1}^n |A_i|$ configurations, where n is the number of parameters and A_i is the set of values for the i -th parameter.

The *simulator* module is responsible for running each experiment, corresponding to a combination of input parameters. To provide reliable estimates of each library’s performance, it applies two *fixes*. First, it attaches the publisher and subscriber threads to isolated cores to avoid interference from other processes. Second, it repeats the benchmarking experiment R times and averages the results. If multiple subscribers are involved, each producing different results, then the simulator first averages the results of the different subscribers, producing the result for a single run. Then, it averages over the results of multiple runs, producing the results for the given input

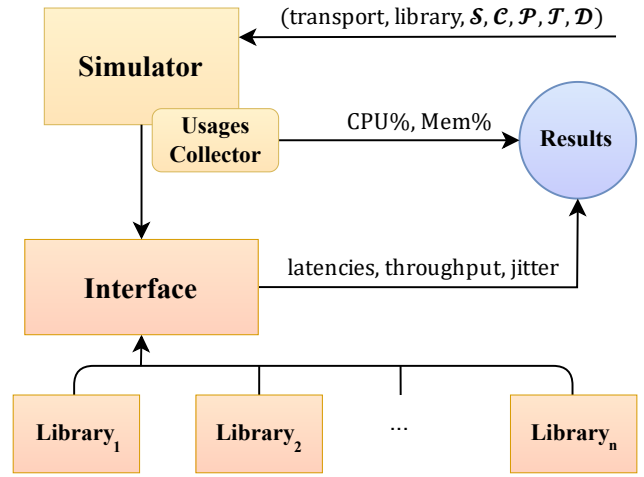


Fig. 1: Overview of the benchmarking suite.

parameters. Furthermore, using the process identifiers of the threads running, the simulator implements the logic to sample the CPU and memory usage percentages at discrete time steps, as described in Section IV-B.

Finally, an *interface* is built as an abstraction to the libraries’ implementations of the basic operations (e.g., bind, connect, send, receive). The interface implements the structure of the benchmarking experiment described in Section IV-C. Any primitive operation is then implemented individually for each different library considered, inside a library-specific file. As a result, to add a messaging library to our suite, it is sufficient to implement the basic functionalities required by the interface. Algorithm 2 provides an abstract overview of our suite.

V. PERFORMANCE COMPARISON

In this section, we present the results of the performance analysis of ZeroMQ, NanoMsg, and NNG.

A. Experimental Setup and Baseline Configuration

We run the benchmarking suite on a machine with the following specifications: Intel(R) Xeon(R) w3-2435 CPU with 8 cores at 3.1 GHz, and 64 GB of RAM.

ZeroMQ, NanoMsg, and NNG are tested for the In-Process, Inter-Process, and TCP transports. In our benchmarks, we always run the publisher and subscribers on the same machine. The number of messages sent is fixed to $\mathcal{C} = 5000$, and the starting delay of the publisher is set to $\mathcal{D} = 1000$ milliseconds. We fix these parameters as they do not show significant variations in the performance of the libraries. We focus on three key metrics: average latency, throughput, and CPU usage. To assess them, we conduct two main experiments — one varying the message size, the other varying the number of subscribers. When varying the size of messages, we fix the number of subscribers to $S = 1$ and vary the message size P in power-of-two values, from 1 KB to 512 KB. When varying the number of subscribers, we discuss scalability by fixing the message size to 32 KB and changing the number of subscribers

Algorithm 1 Publisher (left) and subscriber (right) pseudocodes, for a single run of our benchmark.

1: function PUBLISHER($\mathcal{D}, \mathcal{C}, \mathcal{P}, \mathcal{T}$)	1: function SUBSCRIBER(\mathcal{C})
2: Sleep for \mathcal{D} milliseconds	2: Connect to endpoint
3: Bind to endpoint	3: Initialize timer and array to store latencies
4: for $i \leftarrow 1$ to \mathcal{C} do	4: for $i \leftarrow 1$ to \mathcal{C} do
5: Insert current timestamp into payload	5: Extract timestamp from message payload
6: Fill with 'A's to reach size \mathcal{S}	6: Compute latency using timestamp
7: Publish payload to endpoint	7: Store latency in the array
8: Sleep for \mathcal{T} microseconds	8: end for
9: end for	9: Process and report results
10: end function	10: end function

Algorithm 2 Benchmarking suite pseudocode.

```

1: function BENCHMARKING SUITE( $combs, R$ )
2:   for  $comb \in combs$  do
3:     Initialize metrics cumulative variables
4:     for  $r \leftarrow 1$  to  $R$  do
5:       Run the exp. with parameters  $comb$ 
6:       Attach pub/sub threads to isolated cores
7:       While running, extract (CPU/mem) usages
8:       When done, extract results
9:       Update cumulative variables
10:    end for
11:    Process and report averages for  $comb$ 
12:  end for
13: end function

```

TABLE III: Baseline configurations for the analysis.

Parameter	Configuration		
	Latency	Throughput	CPU Usage
Publishing interval (\mathcal{T})	1000 μ s	0 μ s	0 μ s
Message size (\mathcal{P})		32 KB	
Subscribers (\mathcal{S})		1	
Messages sent (\mathcal{C})		5000	
Publisher delay (\mathcal{D})		1000 ms	

to 1, 2, 4, and 8. For all the experiments, we used $R = 4$ (see Section IV-D).

Given these conditions, we focus on the baseline settings for the publishing interval described in Table III.

We set the publishing interval to $T = 0$ microseconds when measuring throughput, capturing the scenario in which the publisher sends messages at the maximum achievable rate (see Section IV-C). This avoids artificially limiting the throughput, as any non-zero value for T would impose an upper bound on the rate of messages sent. The same value of the publishing interval is used for the CPU usage analysis, as we want to consider the most resource-intensive case, where the publisher is sending messages at the fastest rate. In contrast, for latency analysis, we deliberately use a publishing interval of $T = 1000$ microseconds to decouple the latency measurements from queueing effects and inter-message interference. This ensures the observed latencies primarily reflect the library efficiency.

For all the results we report, each configuration is tested several times, and the results are averaged over the runs. This is done to guarantee a reliable estimate of the performance of the libraries, as detailed in Section IV-D.

In Section V-E, we briefly analyze the other metrics mentioned in Section IV-B. Full results are publicly available [13].

B. In-Process Communication

We analyze the performance of the libraries when the publisher and subscribers run in the same process.

Figure 2 shows the results when fixing the number of subscribers $\mathcal{S} = 1$ and varying the message size, according to the baseline values described in Section V-A. The measured latencies (Figure 2a) range from *less than 10 microseconds* to over one hundred microseconds, mainly depending on the message size. As the message size increases, the average latency also increases, as expected due to the additional processing time required to handle larger messages. In particular, at 128 KB, NanoMsg and NNG exhibit a peak in latency, after which their performance degrades.

Observation 1. *NanoMsg has the lowest latency with small payloads, while ZeroMQ has the lowest latency once payloads become larger.*

Figures 2b and 2c present the throughput and CPU usage results. The throughput increases as the size of the payload increases, reaching a peak at 512 KB. This is expected, as larger payloads can be sent in fewer messages, reducing the overhead associated with message passing. The highest throughput achieved by the libraries ranges *from 3 to 5 gigabytes per second*. In tradeoff with the throughput, we notice a higher CPU consumption as the payload becomes larger, as expected due to the increased amount of bytes to process per time unit. ZeroMQ achieves the highest throughput (≈ 4.8 GB/s). NanoMsg and NNG monotonically increase but peak at lower values, respectively at ≈ 3.8 GB/s and ≈ 3.3 GB/s. Regarding the cumulative median CPU usage (see Section IV-B), we observe that NanoMsg has the lowest CPU consumption for smaller payloads — up to 128 KB — while still maintaining a competitive throughput. However, as the message size increases, ZeroMQ becomes the most efficient library, achieving the highest throughput with the lowest CPU consumption. NNG shows lower throughput with higher latency and CPU usage than the other two libraries.

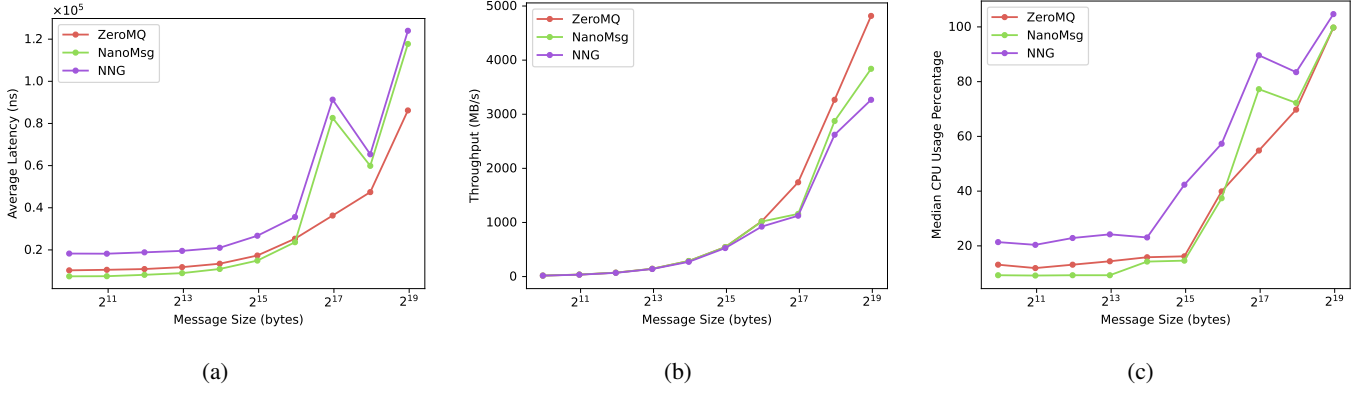


Fig. 2: In-Process Communication: average latency (a), payload throughput (b), and CPU usage (c), when varying message size. The number of subscribers is fixed to 1.

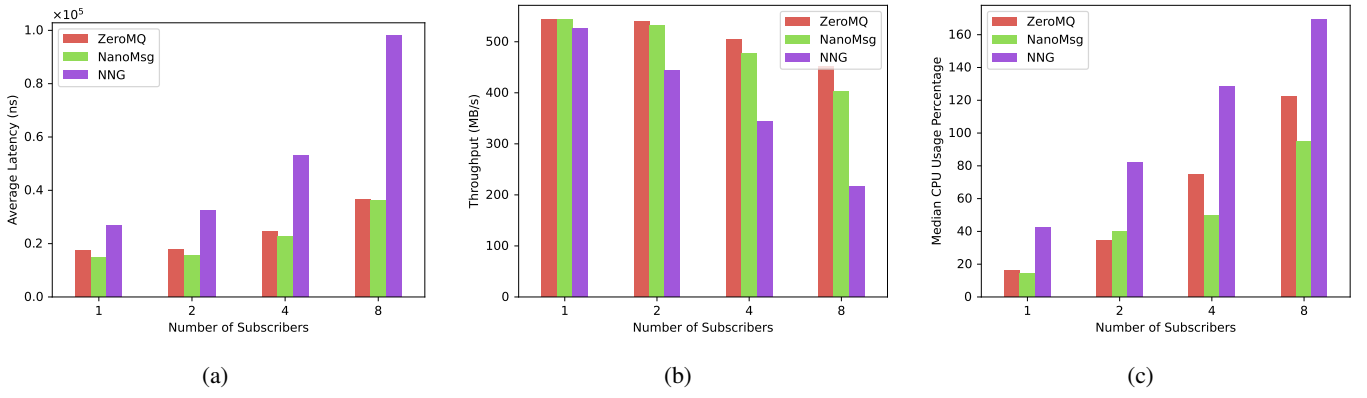


Fig. 3: In-Process Communication: average latency (a), payload throughput (b), and CPU usage (c), when varying the number of subscribers. The message size is fixed to 32 KB.

Observation 2. *NanoMsg shows the best latency, throughput, and CPU results up to 64 KB, after which ZeroMQ overtakes.*

Figure 3, shows the results for the same metrics when the number of subscribers increases. As mentioned, the message size is fixed to 32 KB for these experiments. Figure 3a, shows that latency grows with the number of subscribers, especially for NNG. At the same time, ZeroMQ closes the gap with NanoMsg. We observe similar insights for throughput (Figure 3b), which decreases as the subscribers increase. NNG is competitive for one subscriber, however it does not scale as well as the other libraries. ZeroMQ is the library that scales best, outperforming NanoMsg. The CPU usage result (Figure 3c) highlights that the usage percentage grows as the number of subscribers increases, as expected. The cumulative percentage (see Section IV-B) reaches peaks above 100% for ZeroMQ and NNG, whereas NanoMsg’s CPU usage scales the best for $S = \{4, 8\}$.

Observation 3. *ZeroMQ scales best in terms of throughput, whereas NanoMsg scales best in terms of CPU usage.*

C. Inter-Process Communication

We repeat the same experiments considering the Inter-Process transport, showing the results in Figure 4 and Figure 5.

Figure 4 depicts the results obtained when varying the message size. Average latencies (shown in Figure 4a) range from under $20\mu s$ at 1 KB to over $400\mu s$ at 512 KB, corresponding to 2.5x-4x the In-Process values.

Observation 4. *For Inter-Process communication, NanoMsg achieves the best latency for a message size of up to 128 KB, after which ZeroMQ performs considerably better.*

Throughput results are shown in Figure 4b. Similar to the In-Process results, the libraries reach throughput peaks from 1 to 3 gigabytes per second, with ZeroMQ achieving the highest value (≈ 2.9 GB/s). NNG shows a higher throughput than NanoMsg for the two largest message sizes. The CPU usage results are shown in Figure 4c, indicating that cumulative utilization percentages are up to 1.6 times higher than those observed in the In-Process scenario. ZeroMQ uses less CPU than NNG up to 32 KB, whereas NNG becomes more CPU-efficient for larger payloads.

Observation 5. *For Inter-Process communication, NanoMsg consistently shows the lowest CPU usage for all message sizes.*

Figure 5a shows that, having fixed the message size to 32 KB, as the number of subscribers increases, latency tends to grow. This is more evident for ZeroMQ and NNG, while NanoMsg demonstrates significantly better scalability. In terms

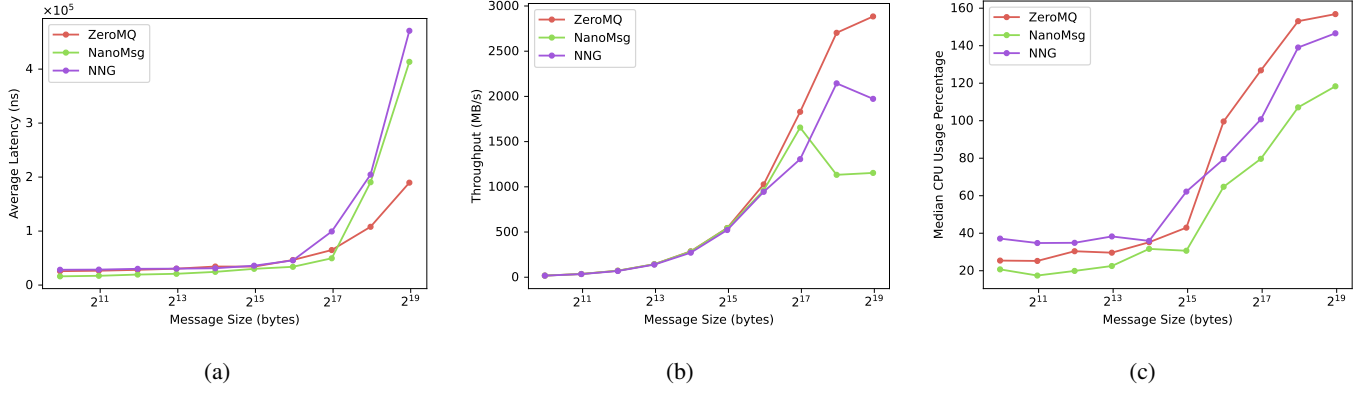


Fig. 4: Inter-Process Communication: average latency (a), payload throughput (b), and CPU usage (c), when varying message size. The number of subscribers is fixed to 1.

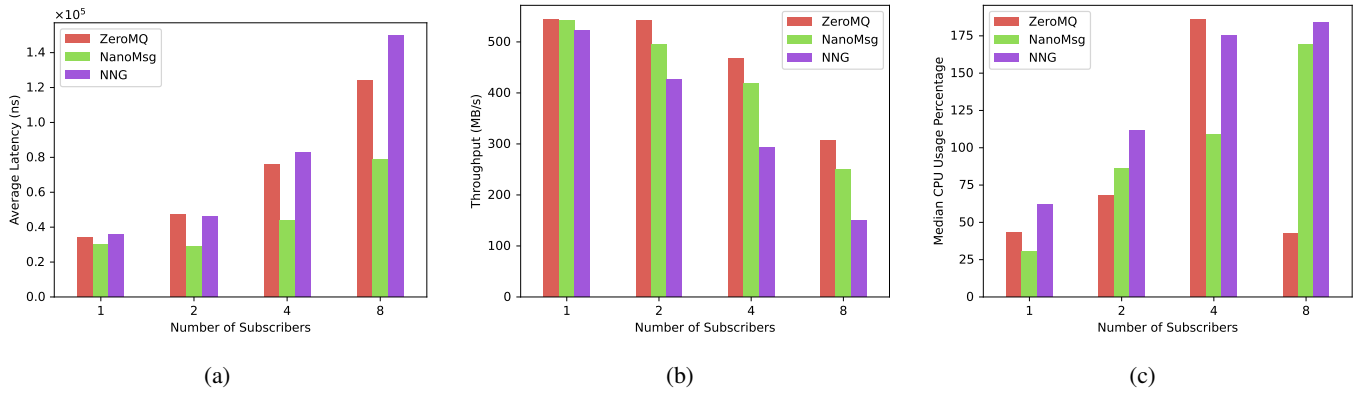


Fig. 5: Inter-Process Communication: average latency (a), payload throughput (b), and CPU usage (c), when varying the number of subscribers. The message size is fixed to 32 KB.

of throughput (Figure 5b), the three libraries show similar performance when only one subscriber is present. However, as the number of subscribers increases, ZeroMQ achieves higher throughput compared to NNG and NanoMsg, with NNG exhibiting a more pronounced decline. Regarding the CPU usage (Figure 5c), the results exhibit non-linear behavior, with each configuration revealing distinct patterns. NanoMsg performs best in scenarios with 1 and 4 subscribers, while ZeroMQ outperforms the others in configurations with 2 and 8 subscribers.

Observation 6. *For Inter-process communication, NanoMsg scales best for average latency, whereas ZeroMQ scales best for throughput. ZeroMQ and NanoMsg, depending on the number of subscribers, are the most CPU efficient libraries.*

D. TCP Communication

This section evaluates the performance of libraries considering TCP-based communication. We replicate the same experiments conducted for In- and Inter-Process communication, configuring the libraries to use the TCP transport. This results in additional overhead due to the traversal of the network stack, even though the messages are routed through the loopback interface. We restrict the TCP experiments to a single host so

that (i) micro-second-level one-way latencies can be measured without requiring sub- μs clock synchronization between distinct machines, and (ii) any variability introduced by network interface cards, cables, or switches is eliminated, allowing us to isolate the overhead of traversing the kernel’s TCP/IP stack.

Analyzing average latency as a function of payload size (Figure 6a), we observe latency values in a range *comparable to those of Inter-Process communication*. This suggests that the overhead of the TCP stack is relatively low. NanoMsg has the lowest latency up to 8 KB, and then exhibits a strange behavior, up to 64 KB, possibly due to an internal limitation of the library. In this region, NNG and ZeroMQ achieve the lowest latency. ZeroMQ becomes the best choice for large message sizes (greater than 128 KB).

Throughput performance (Figure 6b) shows similar results to the one presented for the Inter-Process scenario.

Observation 7. *For TCP communication, ZeroMQ achieves the highest peak throughput (≈ 2.9 GB/s), outperforming NanoMsg and NNG for message sizes higher than 128 KB.*

NanoMsg throughput achieves a peak around 1.5 GB/s for 128 KB but then decreases, while NNG throughput monotonically increases, surpassing NanoMsg for large payloads, and achieving a peak around 2 GB/s. The CPU usage analysis

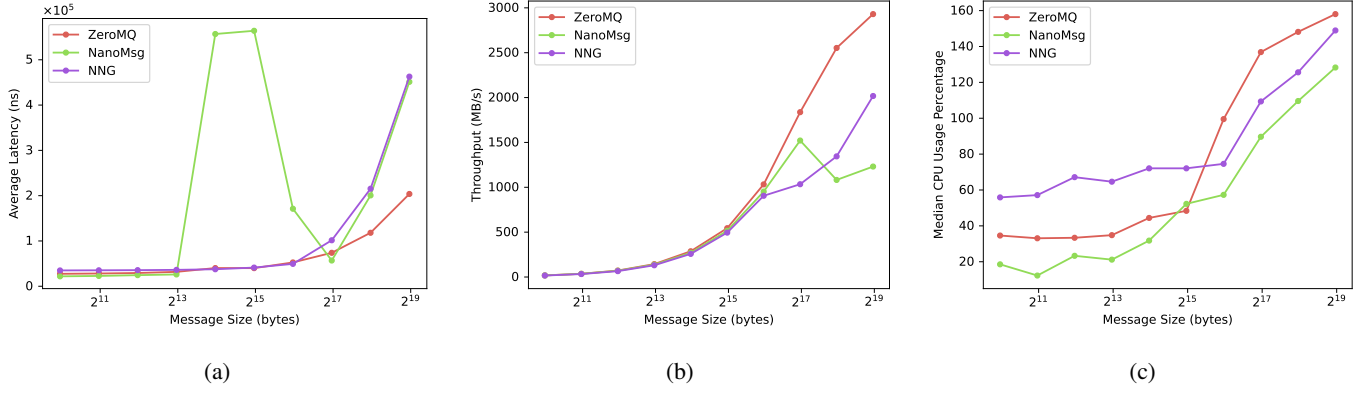


Fig. 6: TCP Communication: average latency (a), payload throughput (b), and CPU usage (c), when varying the message size. The number of subscribers is fixed to 1.

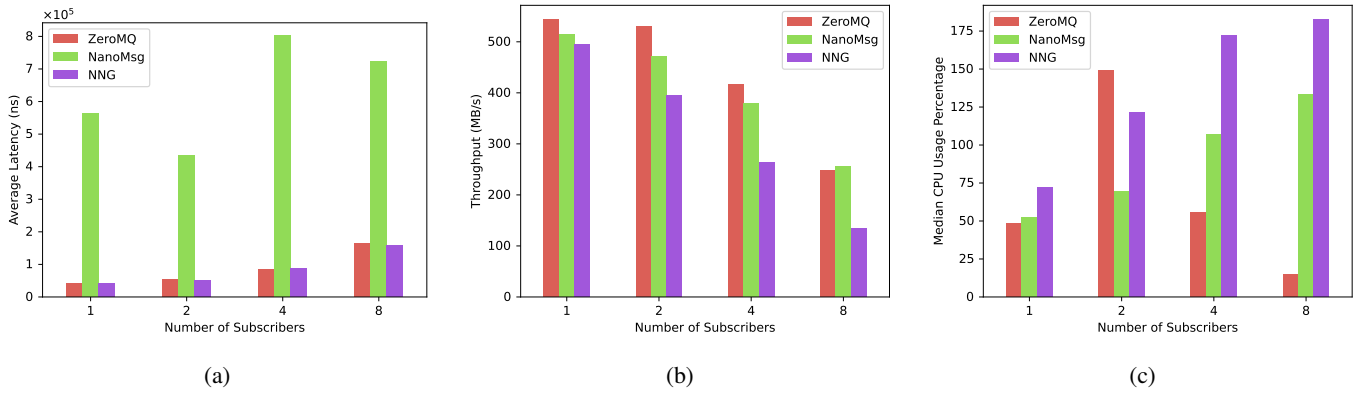


Fig. 7: TCP Communication: average latency (a), payload throughput (b), and CPU usage (c), when varying the number of subscribers. The message size is fixed to 32 KB.

(Figure 6c) indicates that *NanoMsg* generally offers the lowest CPU usage, with an exception at 32 KB where ZeroMQ marginally outperforms it. ZeroMQ and NNG follow, for message sizes respectively lower and higher than 32 KB.

In Figure 7, we examine the impact of scaling the number of subscribers at a fixed payload size of 32 KB. The latency behavior (Figure 7a) shows that the peak in latency exhibited by NanoMsg is also present for a higher number of subscribers. On the other hand, *ZeroMQ* and *NNG* offer relatively scalable latencies across varying subscriber counts. Throughput scalability (Figure 7b) presents a similar trend with respect to the other transports considered. ZeroMQ maintains the highest throughput as the subscriber count increases, except for the scenario with 8 subscribers, where NanoMsg surpasses both ZeroMQ and NNG, providing a higher throughput. Finally, we consider the CPU usage as the subscriber count varies (Figure 7c). As observed in the Inter-Process case, ZeroMQ exhibits non-linear trends. It emerges as the most CPU-efficient library with 1, 4, and 8 subscribers, whereas NanoMsg is more efficient with 2 subscribers.

Observation 8. For TCP communication, as subscriber count grows, *ZeroMQ* and *NNG* scale best for latency, *NanoMsg* for throughput; the most CPU-efficient library is, depending on

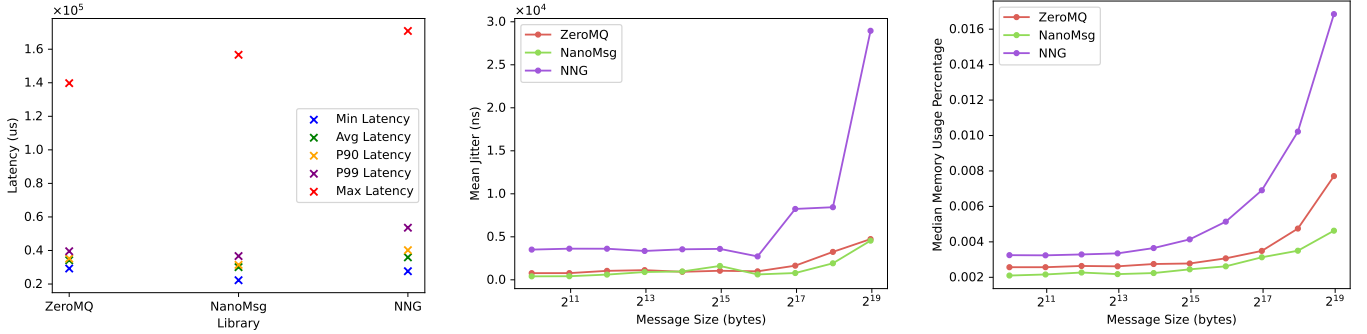
the scenario, either *ZeroMQ* or *NanoMsg*.

E. Other Metrics

The goal of this section is to discuss some other interesting results that we observed during the evaluation. We focus on minimum, maximum, and percentile values for the latency, together with jitter and memory usage. Due to space constraints, we only present the results for the Inter-Process transport; the complete results for every other transport are available in [13].

First, we further discuss latencies by showing more details about their overall distribution over a single configuration. In particular, from the baseline setup detailed in Section V-A, we fix the value of the message size to 32 KB, thus considering a single set of values for the input parameters.

For this setting, Figure 8a shows the minimum, average, p90, p99, and maximum latencies for each library. The plot highlights that, while most latency values (minimum, average, p90, and p99) are close, the maximum latency is significantly higher. ZeroMQ exhibits the narrowest latency distribution, achieving the best result in terms of maximum latency. NanoMsg achieves the best average latency and also performs best for the minimum, p90, and p99 metrics. In contrast, NNG



(a) Distribution of latencies for each library, considering one subscriber, message size fixed to 32 KB, and publishing interval of 1000 μ s. (b) Jitter. We consider one subscriber and the publishing interval fixed to 1000 μ s, varying the message size. (c) Memory usage. We consider one subscriber and the publishing interval fixed to 1000 μ s, varying the message size.

Fig. 8: Inter-Process Communication: results for the jitter, memory usage, and latency values at different percentiles.

shows the widest latency distribution and exhibits the worst results for the maximum latency.

Observation 9. *ZeroMQ minimizes worst-case latency, whereas NanoMsg is best on average results.*

Second, we analyze jitter, using the configuration with a publishing interval of 1000 μ s as presented in Section V-A. The results are shown in Figure 8b. NanoMsg and ZeroMQ exhibit the best performance in terms of jitter, maintaining average variations between consecutive message latencies *in the order of one microsecond*. In contrast, NNG shows a higher jitter for all message sizes, still remaining below ten microseconds on average, except for a peak at the largest payload.

Observation 10. *NanoMsg and ZeroMQ keep the jitter near 1 μ s, whereas NNG shows higher values and spikes for higher payloads, e.g., 512 KB.*

Finally, in Figure 8b, we consider memory usage. We can see that the memory usage for all libraries increases monotonically as the message size increases, as expected. However, all the libraries show *very low memory consumption*, under 0.01% in most cases. In particular, NanoMsg achieves the lowest consumption, followed by ZeroMQ and NNG.

Observation 11. *The memory usage for all three libraries is relatively low, with NanoMsg being the most memory efficient.*

F. Optimality Regions

Finally, we explore every combination of the discrete sets of subscriber number and message size considered so far and identify, for each region of the parameter space, the library that is *optimal*, i.e., performs best according to our figures of merit, *within the scope of the three frameworks evaluated*.

The In-Process communication scenario is shown in the top row of Figure 9. In the latency subplot (Figure 9a), we observe that NanoMsg offers the lowest latency for small to medium message sizes (up to 32 or 64 KB) across the various subscriber scenarios. However, ZeroMQ performs better at larger message sizes, confirming better outcomes in handling larger payloads. Regarding throughput (Figure 9b), ZeroMQ performs overall best, delivering higher throughput across all

configurations of message sizes and subscribers, except some of the ones involving one subscriber or very small payloads (up to 4 KB). For CPU efficiency (Figure 9c), again, NanoMsg often performs optimally up to the message size of 32 KB, from where ZeroMQ usually proves to be more efficient.

The Inter-Process communication scenario is shown in Figure 9, middle row, where similar insights are observed. In the latency subplot (Figure 9d), NanoMsg outperforms the other libraries for almost all message sizes and subscriber combinations, with ZeroMQ being preferable only at the very largest message sizes (256 and 512 KB).

Observation 12. *In-Process and IPC transports present message size thresholds at which latency optimality shifts.*

For throughput (Figure 9e), ZeroMQ is optimal across most configurations, except for regions involving small payloads and a single subscriber. NanoMsg is the most CPU-efficient library in most of the Inter-Process configurations (Figure 9f), with a few exceptions where it is outperformed by ZeroMQ.

Finally, the TCP communication scenario is reported in the bottom row of Figure 9. The latency subplot (Figure 9g) shows a more complex scenario, where NNG becomes relevant, as also mentioned in Section V-D. The three libraries all have regions of optimal performance. In terms of throughput (Figure 9h), ZeroMQ performs well, delivering the highest throughput consistently for almost all scenarios tested. For CPU usage (Figure 9i), NanoMsg is again most efficient for nearly all the tested combinations, except at the largest message sizes (256 and 512 KB), where NanoMsg or NNG slightly surpasses ZeroMQ in CPU efficiency.

Observation 13. *Across all transports and most configurations, ZeroMQ delivers the highest throughput, whereas NanoMsg achieves the best CPU efficiency.*

VI. RELATED WORK

Several studies examine messaging systems, mainly focusing on brokered architectures, with limited attention to brokerless systems. [39], [40] review the state of the art of messaging systems, identifying brokerless libraries as emerging solutions.

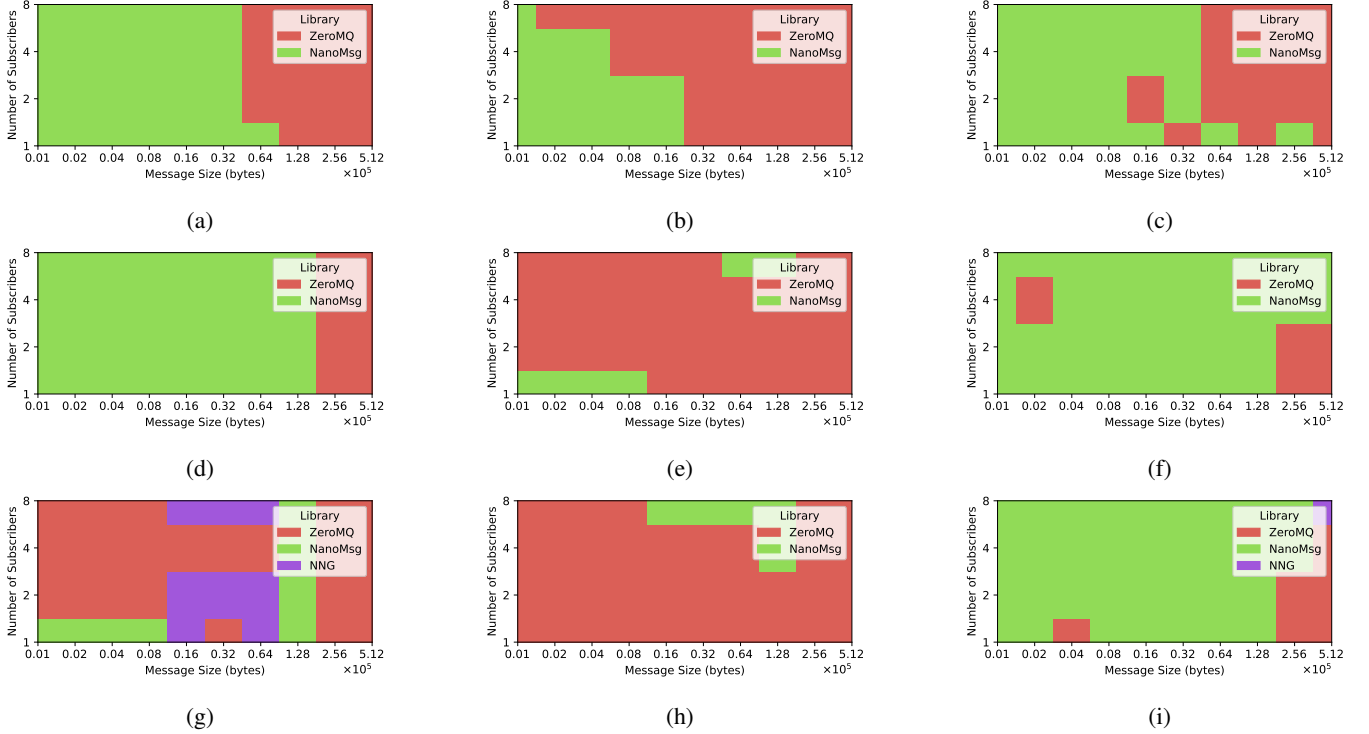


Fig. 9: Regions of optimal performance for the three brokerless libraries evaluated. Rows correspond to the transport mechanism — In-Process (top), Inter-Process (middle), and TCP (bottom), while columns report, from left to right, latency, throughput, and median CPU usage results. Within each panel, the shaded region indicates the library that attains the best result for every combination of message size and subscriber count explored with our tests.

Qualitative comparisons, highlighting ease of installation, documentation quality, and community support, are presented in [4], [5], [11]. Quantitative evaluations of brokered systems are provided in [5], [10]–[12], [40]. [11] is particularly relevant because the authors develop their own testing framework to ensure a fair evaluation. Although a few works [2]–[4] include brokerless libraries among their candidates, they do not focus exclusively on such systems, instead evaluating them alongside brokered solutions or in specialized application contexts. In [1], the authors evaluate Ice, ZeroMQ, and YAMI4 against specific workload and latency requirements in a PUB/SUB scenario, identifying ZeroMQ as the best-performing library, particularly due to its superior scalability with increasing numbers of subscribers. In [3], ZeroMQ, NanoMsg, Asio, FairMQ, and O2 are tuned and benchmarked for a system transmitting terabits per second of data. The study analyzes network and memory throughput as a function of block size and system CPU usage, identifying Asio [41] as the best solution. In [2], the authors evaluate RabbitMQ, ActiveMQ, ZeroMQ, and NanoMsg for transmitting binary JSON data in distributed Computer Numerical Control (CNC) systems, concluding that NanoMsg is the best solution in terms of bandwidth, API flexibility, and POSIX compatibility.

To the best of our knowledge, no existing study includes NNG as a candidate, nor provides a dedicated, comprehensive benchmark specifically for brokerless messaging frameworks.

This gap in the literature is filled by our contribution.

VII. CONCLUSION AND FUTURE WORK

In this work, we presented a qualitative study and a systematic performance evaluation of brokerless messaging libraries. First, we conducted a qualitative analysis based on licensing, documentation, and community support, which led us to select ZeroMQ, NanoMsg, and NNG as the most promising candidates. Then, we developed and released a comprehensive benchmarking suite to evaluate latency, throughput, jitter, CPU usage, and memory consumption across different transports and workload conditions. Our framework is publicly available [13], allowing reproducibility and extensibility.

Our results suggest that selecting the most efficient library largely depends on the specific workload and configuration, highlighting important trade-offs between the libraries; specifically, ZeroMQ excels in throughput and offers the best performance at large payloads, whereas NanoMsg performs best for small message sizes and demonstrates consistent strength in CPU efficiency. Except for some specific configurations, NNG performance is typically less competitive. However, it usually falls within the same order of magnitude. Moreover, NNG is the only library under active development.

In future work, we plan to extend our suite to include multi-machine communication, additional patterns, and libraries.

REFERENCES

- [1] A. Dworak, F. Ehm, P. Charrue, and W. Sliwinski, "The new cern controls middleware," in *Journal of Physics: Conference Series*, vol. 396, no. 1. IOP Publishing, 2012, p. 012017.
- [2] M. Y. Afanasev, Y. V. Fedosov, A. A. Krylova, and S. A. Shorokhov, "Performance evaluation of the message queue protocols to transfer binary json in a distributed cnc system," in *2017 IEEE 15th International Conference on Industrial Informatics (INDIN)*. IEEE, 2017, pp. 357–362.
- [3] V. C. Barroso, U. Fuchs, and A. Wegrzynek, "Benchmarking message queue libraries and network technologies to transport large data volume in the alice o system," in *2016 IEEE-NPSS Real Time Conference (RT)*. IEEE, 2016, pp. 1–5.
- [4] S. Patro, M. Potey, and A. Golhani, "Comparative study of middleware solutions for control and monitoring systems," in *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)*. IEEE, 2017, pp. 1–10.
- [5] E. Bertrand-Martinez, P. Dias Feio, V. d. Brito Nascimento, F. Kon, and A. Abelém, "Classification and evaluation of iot brokers: A methodology," *International Journal of Network Management*, vol. 31, no. 3, p. e2115, 2021.
- [6] B. Goertzel, D. Hanson, and G. Yu, "A software architecture for generally intelligent humanoid robotics," *Procedia Computer Science*, vol. 41, pp. 158–163, 2014.
- [7] V. Andreev, K. Kirsanov, P. Pletenev, Y. V. Poduraev, V. Pryanichnikov, and E. Prysev, "Technology supervisory control for mechatronic devices via the internet," *Procedia Engineering*, vol. 100, pp. 33–40, 2015.
- [8] K. Kirill, "Software architecture of control system for heterogeneous group of mobile robots," *Procedia Engineering*, vol. 100, pp. 278–282, 2015.
- [9] P. Sommer, F. Schellroth, M. Fischer, and J. Schlechtendahl, "Message-oriented middleware for industrial production systems," in *2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)*. IEEE, 2018, pp. 1217–1223.
- [10] D. L. de Oliveira, A. F. d. S. Veloso, J. V. Sobral, R. A. Rabêlo, J. J. Rodrigues, and P. Solic, "Performance evaluation of mqtt brokers in the internet of things for smart cities," in *2019 4th International Conference on Smart and Sustainable Technologies (SpliTech)*. IEEE, 2019, pp. 1–6.
- [11] G. Fu, Y. Zhang, and G. Yu, "A fair comparison of message queuing systems," *IEEE Access*, vol. 9, pp. 421–432, 2020.
- [12] R. Maharjan, M. S. H. Chy, M. A. Arju, and T. Cerny, "Benchmarking message queues," in *Telecom*, vol. 4, no. 2. MDPI, 2023, pp. 298–312.
- [13] L. La Corte, "Messanging libraries benchmark," <https://github.com/hitachienergy/messaging-libraries-benchmark>.
- [14] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM computing surveys (CSUR)*, vol. 35, no. 2, pp. 114–131, 2003.
- [15] J. Kreps, N. Narkhede, J. Rao *et al.*, "Kafka: A distributed messaging system for log processing," in *Proceedings of the NetDB*, vol. 11, no. 2011. Athens, Greece, 2011, pp. 1–7.
- [16] J. Williams, *RabbitMQ in action: distributed messaging for everyone*. Simon and Schuster, 2012.
- [17] ZeroMQ community, "libzmq Licence (Mozilla Public License 2.0)," <https://github.com/zeromq/libzmq/blob/master/LICENSE>, 2025, accessed 22 Apr 2025.
- [18] Martin Sustrik & nanomsg contributors, "nanomsg COPYING (MIT licence with trademark notice)," <https://github.com/nanomsg/nanomsg/blob/master/COPYING>, 2025, accessed 22 Apr 2025.
- [19] Staysail Systems, Inc. and Capitar IT Group BV, "NNG LICENSE.txt (MIT licence)," <https://github.com/nanomsg/nng/blob/main/LICENSE.txt>, 2025, accessed 22 Apr 2025.
- [20] ZeroRPC project, "Zerorpc license (mit)," <https://github.com/0rpc/zerorpc-python/blob/master/LICENSE>, 2015, accessed: 2025-04.
- [21] "Zeromq documentation," <https://zguide.zeromq.org/>, accessed: 2025-04.
- [22] "Nanomsg documentation," <https://nanomsg.org/documentation.html>, accessed: 2025-04.
- [23] G. D'Amore, *Nng reference manual*. Staysail Systems, Inc., 2018.
- [24] "Yami4 documentation," <http://inspirel.com/yami4/book/>, accessed: 2025-04.
- [25] ZeroMQ community, "zeromq/libzmq: Zeromq core library (c++)," <https://github.com/zeromq/libzmq>, 2025, accessed: 2025-04.
- [26] NanoMsg project, "nanomsg/nanomsg: Nanomsg messaging library," <https://github.com/nanomsg/nanomsg>, 2025, accessed: 2025-04.
- [27] NNG project, "nanomsg/nng: Nng next-generation nanomsg," <https://github.com/nanomsg/nng>, 2025, accessed: 2025-04.
- [28] ZeroRPC project, "0rpc/zerorpc-python: Zerorpc - reliable, high-performance communication between distributed processes," <https://github.com/0rpc/zerorpc-python>, 2025, accessed: 2025-04.
- [29] P. Hintjens, *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, 2013.
- [30] "About nanomsg," <https://nanomsg.org/>, accessed: 2025-04.
- [31] M. Sustrik, "Nanomsg rationale," <https://250bpm.com/blog/4/>, accessed: 2025-04.
- [32] "Nng," <https://nng.nanomsg.org/>, accessed: 2025-04.
- [33] G. D'Amore, "Nng rationale," <https://nng.nanomsg.org/RATIONALE.html>, accessed: 2025-04.
- [34] "Zerorpc," <https://www.zerorpc.io/>, accessed: 2025-04.
- [35] M. Sobczak, "Yami4 vs. zeromq," http://www.inspirel.com/articles/YAMI4_vs_ZeroMQ.html, 2013, accessed: 2025-04-26.
- [36] C. Demichelis and P. Chimento, "IP Packet Delay Variation Metric for IP Performance Metrics (IPPM)," RFC 3393, Nov. 2002, standards Track. [Online]. Available: <https://datatracker.ietf.org/doc/html/rfc3393>
- [37] G. Rodola *et al.*, "psutil: Cross-platform lib for process and system monitoring in python," <https://github.com/giampaolo/psutil>, accessed: 2025-04.
- [38] I. Dalmasso, S. K. Datta, C. Bonnet, and N. Nikaein, "Survey, comparison and evaluation of cross platform mobile application development tools," in *2013 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*. IEEE, 2013, pp. 323–328.
- [39] S. Celar, E. Mudnic, and Z. Seremet, "State-of-the-art of messaging for distributed computing systems," *Vallis Aurea*, vol. 3, no. 2, pp. 5–18, 2017.
- [40] J. Yongguo, L. Qiang, Q. Changshuai, S. Jian, and L. Qianqian, "Message-oriented middleware: A review," in *2019 5th International Conference on Big Data Computing and Communications (BIGCOM)*. IEEE, 2019, pp. 88–97.
- [41] "Asio c++ library," <https://think-async.com/Asio/>, accessed: 2025-04.